SAND97–2519
Unlimited Release
Printed October 1997

Distribution
Category UC–905

Experiences Implementing the MPI Standard on Sandia's Lightweight Kernels

Ron Brightwell David S. Greenberg
Computational Sciences, Computer Sciences, and Mathematics Center
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1110

Abstract

This technical report describes some lessons learned from implementing the Message Passing Interface (MPI) standard, and some proposed extensions to MPI, at Sandia. The implementations were developed using Sandia-developed lightweight kernels running on the Intel Paragon and Intel TeraFLOPS platforms. The motivations for this research are discussed, and a detailed analysis of several implementation issues is presented.

Acknowledgment

Appreciation is extended to the following people for contributing to this research: Lee Ann Fisk, Tramm Hudson, Arthur B. Maccabe, Kevin McCurley, Lance Mumma, Rolf Riesen, Lance Shuler, David van Dresser, and Stephen Wheat. In addition, the authors would like to thank Jeff Brown, Pang Chen, and David Womble for useful discussions.

Contents

Introduction **6**

Background **6**
MPI . . . 6
MPI-2 . . . 7
SUNMOS . . . 7
Puma . . . 7
MPICH . . . 7
MPICH ADI Design . . . 8
MPICH ADI-1 Design . . . 8
MPICH ADI-2 Design . . . 8

Initial Design and Implementation **9**
Prototyping with SUNMOS . . . 9
Initial Puma Design . . . 10
Evaluation of Initial Design . . . 10

Portals Design and Performance in MPI **12**

Puma Design and Performance in MPI **13**
Progress . . . 13
Using the Message Co-processor . . . 14
Design for Thread Support . . . 14

Beyond Sends and Receives **15**
Collective Operations . . . 15
One-Sided Communications . . . 15
Combined Block Issues . . . 16
Real-Time Channels . . . 17

Validating MPI **17**

Conclusion **19**

References **20**

Figures

1 Paragon Pingpong Latency . . . 12
2 TeraFLOPS Pingpong Latency . . . 13

Experiences Implementing the MPI Standard on Sandia's Lightweight Kernels

Introduction

Sandia National Laboratories, and other institutions requiring highest-performance computation, have been using massively parallel processing (MPP) techniques for over a decade to harness thousands of microprocessors. The key to the use of MPPs is that many problems can be decomposed (often along domain relevant boundaries) into subproblems which can be assigned to individual processors. Informational dependencies between pieces are resolved by sending data between processors over a dedicated, high-speed, interconnection network.

Through multiple generations of machines and multiple vendors (such as nCUBE, TMC, Intel, and Cray) a stylized version of communication developed. One node,[1] the sender, packages data into a *message* and sends it over the interconnect to a second node, the receiver, which unpacks the message. Each vendor implemented the packing of data, method of describing destination, buffering of data, method of using the interconnect, method for recognizing message arrival, method of acknowledgment, etc., to best match each specific hardware and each perception of user requirements.

In order to insulate application programs from the peculiarities of each new system, message passing libraries or wrappers were created. Eventually in 1993, a forum was set up to create a message passing standard. In 1994, the Message Passinge Interface standard [9] was published.

However, the real work had just begun. MPI specifies a high-level interface suitable for being ported to a wide variety of systems, from large-scale, tightly integrated MPPs to small clusters of workstations. Yet in the MPP world, MPI was expected to do much more – it had to replace highly tuned, highly integrated vendor systems which were often an integral part of the system software. This paper describes experiences in decoupling the user interface from the system software.

In developing Sandia's MPI implementation, the work of two ongoing projects was leveraged: the Puma operating system [12] work at Sandia and the MPICH [4] work at Argonne National Laboratory and Mississippi State University. Puma provided the basic operating system services required for internode data movement, and MPICH provided a layered approach to MPI which allows implementors to concentrate on data movement. This layered approach also allowed for the lessons learned from this implementation to be fed back to both development groups in order to better define the roles and abilities of each layer.

Background

MPI

The Message Passinge Interace standard [9] was developed over a 12 month period in 1993-1994 through meetings involving more than 80 participants from about 40 different organizations. The MPI specification document was published in 1994.

The core of MPI is the functions and semantics that define data movement. In order to avoid mandating any specific implementation behavior in regard to message buffering, the standard defines four different semantics associated with sending a message. Completion of a send operation is local to the sending process if no action from the destination process is needed to complete the operation. Likewise, completion of a send operation is non-local if some interaction with the destination process is required.

The most basic send mode is standard mode. Completion of a standard mode send may be either local or non-local, depending upon the implementation. For example, implementations that want to avoid buffering

[1]Over time single processors have been replaced with multiple processor nodes.

unexpected messages (i.e. those messages for which no receive had been posted prior to the arrival of the message) at the receiver may choose to implement standard sends via a rendezvous protocol.

Completion of a synchronous mode send is non-local. A synchronous send completes only when a matching receive operation has begun at the receiver. This mode provides a rendezvous point between the sender and the receiver.

Completion of a ready mode send is local. A ready send is an opportunity to take advantage of a hint from the application about the state of the receiver. Use of a ready send by the application guarantees that a matching receive has been pre-posted at the receiver, in hopes that this will eliminate any handshaking or protocol overhead needed to handle unexpected messages.

Completion of a buffered send is local. In the buffered send mode, the application explicitly provides buffer space to the MPI implementation to use for copying local send buffers. Buffered sends can complete regardless of the state of the receiver as long as there is sufficient space within the user-provided buffer in which to copy outgoing messages.

Because there are multiple send modes and only one receive operation, the receive operation must be implemented to respond with the correct protocol to handle the send-side semantics. Therefore, the initiation of an MPI send operation must always provide protocol information so the receiver can respond appropriately.

The MPI standard also defines a rich set of collective operations, such as broadcast and reduction operations, which can be layered on top of the MPI point-to-point functions.

MPI-2

Since the release of the MPI standard document in June of 1994, research in the area of message passing continued. Researchers proposed some extensions to the original interface and semantics to provide more robust communication and possibly higher performance. In March of 1995, the MPI forum began meeting to discuss corrections and additions to the MPI standard. As a result, the MPI-2 standard [10] was released in July of 1997 and contained definitions for process creation and management, one-sided communications, extended collective operations, external interfaces, and parallel I/O.

The definition of an interface for doing one-sided communications is an attempt to achieve higher performance by eliminating the protocol overhead necessary to achieve MPI-1 peer communication semantics. Rather than send and receive buffers, the application opens up a "window" to which remote processes can put or get data without any explicit interaction from the local process.

SUNMOS

SUNMOS (Sandia/University of New Mexico Operating System) [8] was designed to better take advantage of the available compute and communication power of the nCUBE and Intel Paragon machines, while maintaining a small footprint to leave as much memory as possible for application use. SUNMOS is a single-tasking lightweight kernel optimized for message passing architectures, and does not contain a majority of the functionality and constructs present in most standard, full-blown UNIX operating systems.

Puma

As its successor, the Puma operating system attempts to build off of the experiences of SUNMOS, to provide a more flexible, lightweight, high performance message passing environment for massively parallel computing. Puma has been developed for the i860-based Intel Paragon platform, and has been ported to the Pentium-Pro-based Intel TeraFLOPS platform.

Message passing in Puma is accomplished through the use of *portals*, which are openings in a process's address space into which the Puma kernel can deposit incoming messages. See [12] for a complete description of the different types and options of Puma portals.

MPICH

MPICH [4] is a portable implementation of MPI developed jointly by Argonne National Laboratory and Mississippi State University. MPICH contains an abstract device interface (ADI) software layer which allows it to be ported to any message passing or shared memory platform. The ADI pinpoints the device specific functionality needed to implement MPI and greatly reduces the amount of effort required to port MPICH. Because MPICH tracked the activity of the MPI Forum and was the first working implementation available at the time the standard was published, it established a foothold with vendors and others porting MPI to specific platforms. MPICH is currently the most widely used implementation of MPI.

MPICH ADI Design

MPICH is designed so that there are two layers. The first is an architecture-independent layer which contains all of the MPI API functions which do not depend on any communication. Included in this layer are the functions for virtual topologies, datatype management, error managment, and so forth. Also included in this layer are those functions which may utilize message passing, but which can be layered upon MPI communication. The second layer, or ADI, is an architecture-dependent layer which is responsible for performing communication and data movement operations. The ADI is a low-level interface upon which a message passing layer, such as MPI, can be built. The ADI functions contain entry points for starting, testing, and completing message passing operations.

MPICH ADI-1 Design

The first ADI implementation (ADI-1) [3] contains approximately 30 functions for communication with a device implementation. The ADI communicates to the device through these functions, and the device communicates information back to the upper layers through device handles. These handles are structures which contain the state of a particular communication request. Certain implementations may not need to implement all of these functions, and all may be defined as macros to increase performance or remove them from compilation.

The send handle is also separated into two parts. The first part is device-independent and contains information pertinent to most message passing systems, such as destination process, number of bytes, and operation completed flag. The second part is device-dependent, allowing for information to be provided for a specific device.

Since MPI has non-blocking communication, posting an operation simply starts it. Once an operation is started, it makes progress through ADI functions which perform a non-blocking test for completion or a blocking completion operation.

MPICH ADI-2 Design

The second ADI implementation (ADI-2) [5] attempts to provide more functionality for multi-protocol and multi-device implementations. Multi-protocol implementations can adjust the protocol based on size of message, machine hardware structure, type of message, or any other property which might change either the quality of service needed for the message or the characteristics of the network. Multi-device implementations are a specific way of encapsulating a set of protocols together. Although ADI-1 did not provide explicitly for a multi-protocol device, most implementations provided a two-level protocol to lower latency for short messages and increase bandwidth for longer messages. For example, an ADI function could be defined as a macro which examined message length and then called a short send or long send function, depending on the number of bytes to be transfered. ADI-2 assumes multiple levels of protocol and is designed to allow an implementation to provide this information. ADI-2 also allows for a multi-device implementation. One of the shortcomings of ADI-1 was that it was not well designed to handle platforms where there are several ways of intermixing message passing. For example, on a network of workstations, it is desirable to use shared memory for message passing between processes on the same workstation and a different method for inter-workstation communication. The design of ADI-1 did not prohibit such a device, but ADI-2 was designed with such implementations in mind.

ADI-2 was also designed to limit the amount of overhead in some of the ADI calls. For example, a blocking send request in ADI-1 would build a request argument and fill it with the information necessary to complete the operation. ADI-2 attempts to optimize this operation by avoiding the construction of the request and just providing the necessary information through function parameters. ADI-2 also provides the capability for a device to provide information about the operations necessary to start, progress, and complete operations. Function pointers have been added to the request handles so that the device can choose the necessary operations. Whereas in ADI-1 all operations were completed by calling the same function, ADI-2 allows the device to specify the function that must be called to complete an operation. This allows the device to construct a state machine using function pointers kept inside a request rather than storing the state inside the request and implementing the state machine several times within multiple functions.

Initial Design and Implementation

The goal of the MPI Forum was to provide an interface that was not only usable for application developers, but that was also able to achieve high performance from the underlying message passing hardware. Similarly, the goal of the the SUNMOS and Puma operating systems was to provide the highest possible message passing performance to portable libraries. Thus, the implementation of MPI on SUNMOS and Puma allowed for verification as to whether the goals of both MPI and the operating systems were met.

Prototyping with SUNMOS

Using either low-level SUNMOS primitives or an emulation of Intel's NX message passing library, users can achieve 160 megabytes per second bandwidth for messages greater than 30 kilobytes and zero-length message latencies as low as 17 microseconds using the message co-processor [7]. The initial goal was for an MPI library was to demonstrate at least 90% of the achievable bandwidth and no more than twice the latency.

The first step in testing SUNMOS' ability to efficiently host MPI was to port the MPICH code onto Sandia's paragons. Since the MPICH distribution already included an implementation layered on top of NX this was quite straightforward.

The next step was to replace the MPICH ADI routines for send and recv with new routines which used the SUNMOS low-level primitives called `_nsend` and `_nrecv`. These routines provided a simple matching mechanism which could be used to let the system software place incoming messages into appropriately prepared user buffers rather than into general, global communication buffers. This avoidance of buffer copying is the essence of how SUNMOS provides high bandwidth, and thus it was important to have MPI make use of the matching.

Unfortunately, MPI required slightly more sophisticated matching than SUNMOS provided. MPI matches a source identifier, a user-defined type, and a context from incoming messages with corresponding information supplied by pre-posted receives. The source identification match was easily mapped into SUNMOS's matching of source process id and group id. The user-defined type could map to SUNMOS's tag matching, including the ability to define a wildcard at the receive side. Alternatively, the MPI context could be encoded in the SUNMOS tag. A difficulty arose when both user types and MPI context were mapped to SUNMOS tags. The MPI context can never match a wildcard, while the user type can sometimes match a wildcard. SUNMOS did not provide a way of specifying which portion of the tag could be wildcarded. (As will be seen below one of the extensions added to Puma was to allow a wildcard mask).

The inability to perform all matching at the time of system reception of messages left two alternatives. An approach simliar to the MPICH NX implementation could be used in which every message is buffered, and then matching is done by the MPI library, or an MPI library with restrictions on the matching could be done. Since the focus was on performance, the latter was chosen.

Matching could be restricted in two ways: prohibit wildcarded user types or ignore MPI context. Since many of the applications at Sandia used wildcarded types and almost all applications used only the default MPI context of MPI_COMM_WORLD, context was ignored. During the implementation, it was realized that the context information could be passed in an unused SUNMOS message header field and tested when the MPI library gained control. A mismatched context could then cause a fault, rather than allowing an unsupported function to quietly continue.

When the restricted implementation of MPI was built on the SUNMOS primitives the performance was remarkably good. In fact, it was difficult to distinguish the performance results of using MPI from the results of using the native SUNMOS routines. Several applications transitioned over to using this MPI library on the large Sandia Paragon (Acoma) and have not only run well there, but were easily ported to the Sandia

TeraFLOPS machine (Janus) when it arrived. Thus, MPI served its stated purpose of providing portability and performance when the use of contexts was restricted.

Several questions remained to be resolved, however. The most obvious question was whether the full use of contexts could be included without degrading performance. As is shown below, this turned out to be relatively easy. A more subtle question was how to increase the robustness of performance. The SUNMOS tests and many of the applications designed for the Paragon were carefully tuned to ensure that receives were pre-posted. In addition, messages tended to be large. In order for MPI to be truly successful, it had to handle other cases gracefully. It was hoped that new concepts included in Puma (such as the portal) would allow the creation of an MPI library which met all of Sandia's performance and portability needs.

Initial Puma Design

The initial design of the transport layer for the Puma implementation of MPI was a two-level protocol based on message size. This approach attempts to decrease the latency of small messages while increasing the bandwidth of large messages.

The initial implementation of MPI was layered on top of a user-level portal library. This layered approach was chosen for several reasons. The primary reason was the expectation that applications wishing to make use of portals would do so through the user-level portal library. The user-level portal library was designed to be a buffer that would insulate the developer from the low-level details and inner workings of the portal structures. Another reason was that the MPI library and the operating system were being developed concurrently. Using the user-level portal library helped to decrease the amount of dependence the MPI library had on kernel and associated library structures.

At the time of development, few libraries were actually making use of the user-level portal library, and using it in MPI offered the chance to evaluate its capabilities and performance. MPI and Puma could benefit from each other. However, co-development has an inevitable cost – one effort is often left waiting for the other to provide key features. For example, the sychronous protocol was designed to make use of the acknowledgment feature of some memory descriptors, but the mechanisms that generate acknowledgments had not been implemented when the MPI library was ready to make use of them.

On the positive side, co-development allows early changes in design and implementation. Because the MPI implementation utilized nearly all of the features of all the different types of portal memory descriptors, it became a good tool with which to stress test the portal libraries and kernel. As a result of its use with MPI, the user-level portal library underwent some cosmetic changes in an effort to increase its usability. The API was modified to be more consistent, and some additional functionality was added for a few basic portal manipulation operations. The MPI implementation was adapted to the new interface and continued to be used as a regression tool to insure that message passing functionality within the kernel and libraries continued to work correctly as the operating system evolved.

Evaluation of Initial Design

Initial performance results of MPI were very disappointing. Tests showed MPI zero-length message latencies to be an order of magnitude greater than the lower bound SUNMOS had shown was possible.

In order to find the cause of the poor performance, the MPI library was instrumented, and timings were taken of various operations within MPI. For example, when posting a receive, the implementation acquires the necessary resources, configures the information that is specific to the receive (buffer pointer, length, tags, etc.), checks the unexpected message queue, and either posts or copies the message.

Essentially, the device layer functions were broken down into smaller pieces, each of which performs a specific function, and analyzed. Not only were regions of code instrumented, but counters were added to keep track of data such as the number of messages that were deposited directly into the user buffer, the number of unexpected messages that were buffered, and the average and maximum number of unexpected messages that were queued. All of this data had to be compiled in order to discover the problem areas.

Unfortunately, even this level of instrumentation was not enough. In most cases, the problem areas involved calls to one or more user-level portal library functions. Therefore, the user-level portal library needed to be instrumented to find the real cause of most performance bottlenecks. The user-level and system-level portal libraries were analyzed in order to determine the best way to insert instrumentation.

However, this analysis immediately revealed some problems before any instrumentation occured. Tracing through the various layers of function calls identified several opportunities for optimization.

One optimization was to eliminate duplicated checks. The portals library is divided into routines which each perform a specific function. However, each function validates the user arguments and then converts them in order to access system structures. Since several calls could be needed in order to set up a portal in a desired fashion, the validation checks were unnecessarily repeated.

A second problem identified was the inability of MPI to convey any information through the portal library interface about how the match list and match list entries were being used. Because portals were designed to be flexible enough to handle many different protocols and message passing semantics, the library interface was also designed to be very general. However, this generality prevents performance optimizations tied to specific uses of portal structures.

For example, there are optimizations that MPI can take when activating and de-activating match list entries that the user-level portal library cannot perform. There are three different error or overflow possibilities at each match list entry. Either the incoming message does not match at the current entry (no match), or there is insufficient space in the memory descriptor (no fit), or there is no buffer available at the current entry (no buffer). When a match list entry is de-activated, the portal library traverses the match list, updating each entry's pointers to insure that the inactive entry is not the target of any overflow pointer. However, the semantics of MPI are such that two of these overflow conditions should never occur. A match list entry will always have an available buffer associated with it, so 'no buffer' overflows will never happen. Also, MPI has no concept of truncated messages, where the size of the incoming message is larger than the matching receive buffer. Therefore, a correct MPI program will never use the 'no fit' overflow condition. Since MPI does not make use of these two overflow conditions, de-activating a match list entry is simplified.

The portals library also 'locks' the match list when performing updates. Because the library does not know the structure of the match list, certain steps have to be taken to insure the integrity of the list. In the MPI implementation, there are overflow entries at the end of the list which will insure that a message will always be received if there is no posted match list entry. This allows MPI to avoid using a locking mechanism for additions and deletions to the match list structure. The libraries only manipulate individual entries and there is no way to convey information about the list structure as a whole.

It became apparent that many of performance problems in MPI could be eliminated by consolidating the operations into those which the MPI library would perform often. Because portals are in the application's address space, it was decided to attempt to optimize MPI by eliminating as much dependence as possible on the user- and system- level portal libraries, allowing the MPI library to manipulate the portals structures directly.

The effects of this change on the performance of the library were dramatic. Latencies were reduced by an order of magnitude, within a factor of four of the SUNMOS numbers. Several changes were also made to the manner in which message receipt acknowledgments were handled, in an attempt to use the portals structures more efficiently. While these changes did not directly effect latency numbers, there was a significant increase in the bandwidth performance.

Latency numbers continued to decrease as the operating system was tuned and as support for using the message co-processor was implemented. A more complete and detailed description of the initial implementation of MPI on Puma portals can be found in [2].

Figure 1 shows the latency of several different communication mechanisms for all three operating systems available on the Paragon using the message co-processor. The version of OSF AD/1 used is release 1.4. Sandia has been told that this is the final release of OSF for the Intel Paragons. The version of SUNMOS used is release 1.7.1 which is also intended to be a final release. Both these versions are in active use in production systems. The version of Puma used is informally considered release 0.5 and was essentially orphaned when work moved to the TeraFLOPS machine.

When the OS is kept constant the implementations of MPI tend to be slightly superior to those of NX. Since NX is the primary message passing library for many production applications it has been carefully tuned. Thus it appears that added functionality in MPI does not lead to a loss in performance. The differences in performance between SUNMOS, OSF, and Puma are probably mostly due to differences in amount of time spent tuning the implementations.

Specifically, the timings for SUNMOS exhibit the best performance, except in the case of the MPI library built upon SUNMOS's NX compatibility library. This can be attributed to the quality of the implementation

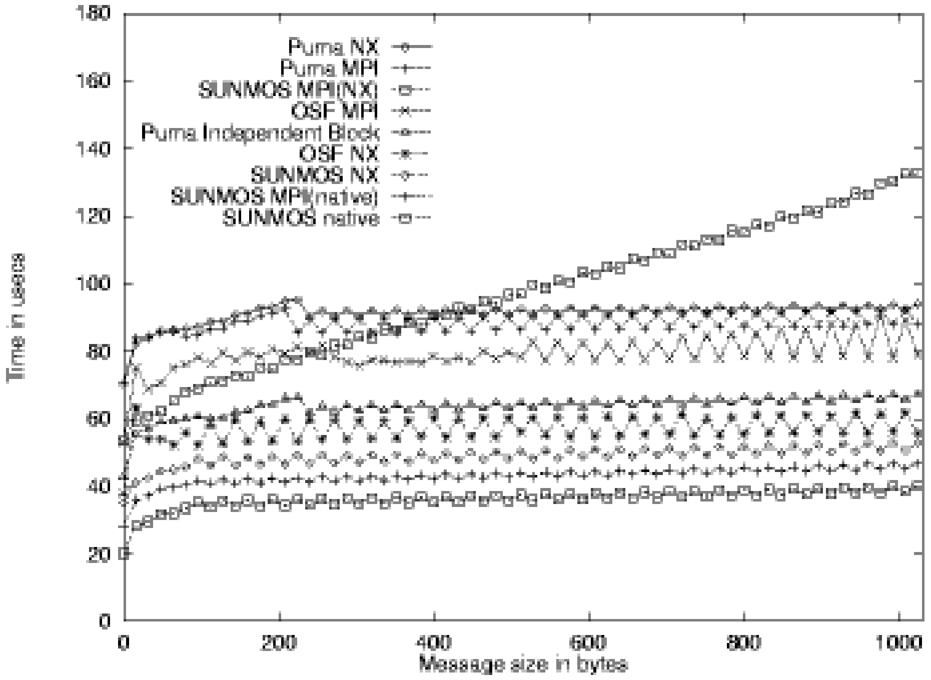

Figure 1: Paragon Pingpong Latency.

of the NX-compatible device in MPICH. Notice that the SUNMOS native MPI implementation outperforms the SUNMOS NX implementation, demonstrating the ability of MPI to achieve high peformance. The communication performance of the Puma independent block memory descriptor is less than that of OSF NX, but the difference between the MPI implementations built on top of each one of those facilities remains somewhat constant. Again, notice for Puma that MPI slightly outperforms the NX implementation.

Figure 2 shows the latency for two of the system level portals and the MPI and NX libraries for Cougar on the TeraFLOPS machine using the message co-processor. These numbers represent what Intel calls release 1.2 WW24, which is a fully functional release but not the final tuned version of Cougar.

It is hard to draw firm conclusions from interim releases, but it appears that the relative overhead of the MPI library on the TeraFLOPS machine is less than that on the Paragon. While the overhead of the MPI library for Puma on the Paragon was almost half of what the independent block memory descriptor could achieve, the same overhead for the TeraFLOPS machine is about one-third.

Also, the same MPI library which slightly outperformed NX for Puma on the Paragon is now slightly slower than NX for Cougar on the TeraFLOPS. This may be a result of improved work on NX or may just be a function of the relative advantages seen by each library in moving from the i860 to the Pentium-Pro processor.

Portals Design and Performance in MPI

Portals are designed to provide all of the communication and memory descriptions needed to build high-level protocols. The main design characteristic of Puma and Portals is the ability of the kernel to deposit messages directly into the user's buffer. The portal match list structure is designed to allow for the context and tag matching capability needed for MPI. The initial implementation of MPI utilizes nearly every feature of portals and the available memory descriptors. The independent block memory descriptor is used for pre-posted receives and acknowledgments. The dynamic block is used to collect unexpected messages, while the single block is used for reading in the long protocol. The acknowledgment and reply features of memory descriptors are also used. The only additional feature added to portals to support MPI was the ability to atomically search the dynamic heap and activate a memory descriptor if the requested message is not found. This allows MPI to search the queue of unexpected messages that have arrived and post a receive if the

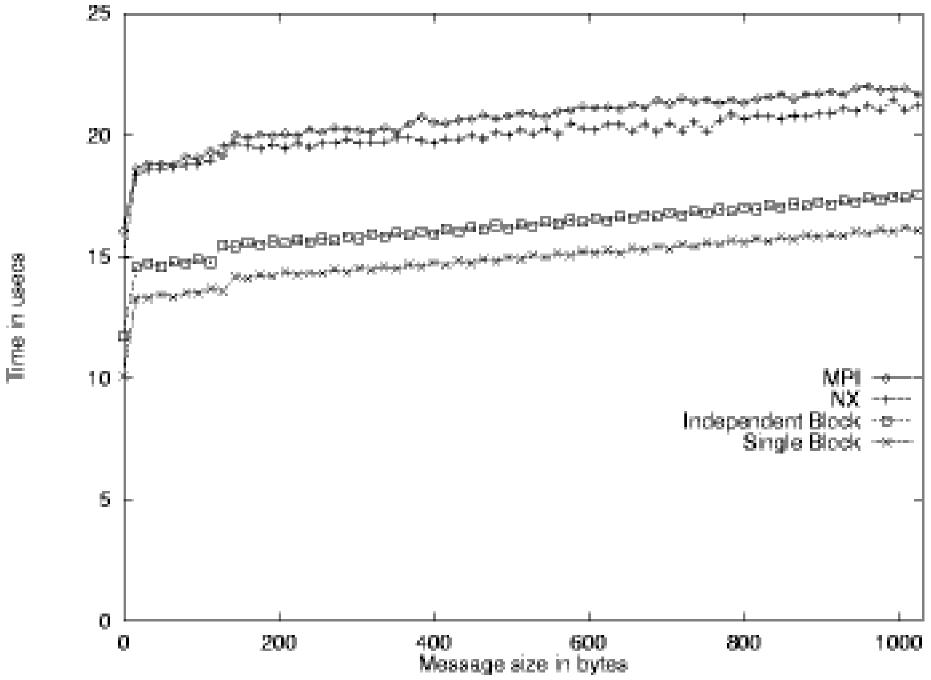

Figure 2: TeraFLOPS Pingpong Latency.

desired message has not arrived.

Since MPI essentially mandates that some protocol information be sent with data messages, use of the fastest performing Puma portal, the single block memory descriptor, cannot be used for pre-posting receives. The single block deposits the data directly into memory without copying any of the message header information. In order to get the header information into user space, the memory has to be validated and the header has to be copied. However, the MPI implementation uses only a few fields in the message header structure. Latency numbers could possibly be improved by eliminating the copying of fields in the message header which the MPI implementation does not use.

MPI will also benefit from the use of the combined block memory descriptor. This descriptor allows pieces of a single incoming message to be deposited at different offsets within a logically contiguous buffer. For non-contiguous datatypes used in point-to-point communication, MPI currently allocates a temporary buffer and copies the data on both the sending and receiving sides. For one-sided communications, data is packed when the target buffer is contiguous, but is sent in individual blocks when the target buffer is non-contiguous. Once implemented in the Puma kernel, the combined block memory descriptor will eliminate the need to pack and unpack data for non-contiguous datatypes. Not only will this reduce the number of memory allocations and memory-to-memory copies, but will also reduce the amount of code complexity needed to handle the special cases of dealing with non-contiguous data. See page 15 for a discussion of the use of combined blocks in one-sided communications, and page 16 for a discussion of combined block implementation issues.

Puma Design and Performance in MPI

Progress

One of the implementation issues for any MPI library is that of progress. Unlike previous message passing interfaces, the MPI standard mandates how and when certain message passing operations complete. In order to insure that MPI operations make progress, most implementations have a progress engine built into the library function calls. That is, regardless of the MPI function the user calls, the implementation attempts to complete any outstanding operations. Because the implementation cannot determine the frequency with which the user application will make calls to the library, some implementations have to use a timer interrupt to insure that operations complete. A *flick* operation such as that used in Intel's NX message passing library was considered in MPI-1, but was rejected for inclusion in the standard.

Portals eliminate the need for any asynchronous progress mechanism. Since the kernel sends and receives messages without any explicit involvement from the user process, it maintains progress for MPI operations without using a timer or an explicit progress mechanism in every library function call.

Using the Message Co-processor

Puma supports the use of multiple processors on a compute node through several different mechanisms [2]. One possibility is to dedicate a processor to message passing. In this mode, the kernel runs on a dedicated processor while the application runs on the remaining processors. This can lead to significantly improved message passing performance since the application need not be interrupted by the kernel in order to handle message passing events.

However, co-processor mode also made the interaction between the kernel and the MPI library more complex. For example, the kernel updates certain structures in the application's address space to signify the arrival of a message on a portal. If the kernel and application are running on the same processor, then the arrival of a message and the updating of these structures guarantees that the entire message has arrived into the application's memory. If the kernel and application are running on separate processors, the kernel may be updating structures as the message is deposited in the application's address space, before the entire message has been received.

For large messages, the notification of message arrival and completion may be signified by the kernel before the last byte of data has been received. The current implementation of the Puma kernel prevents the user from seeing incomplete data by setting a flag to inform the application when the kernel is active. A message is guaranteed to have completely arrived when both the portal structures have been updated and the active flag is clear. However, this solution can overcompensate and cause applications to wait for the kernel to complete the processing of a subsequent message. For time sensitive applications this may be unacceptable and a different approach will be necessary.

The use of the second processor not only for processing message, but also for manipulating MPI library structures needs to be investigated further. In the current implementation, calls to the MPI_WAIT and MPI_TEST family of functions check for message arrival, fill in the MPI_STATUS structure, and free any resources associated with the message. Some of this work could be offloaded to the message co-processor, possibly reducing some time cosuming operations.

For example, when using non-contiguous datatypes, the MPI library allocates space and packs the message into a contiguous buffer. If a portion of the MPI library were running on the co-processor, it could allocate the memory and do the copying while the application continued to run on the other processor, providing more overlap between communication and computation. On the receiving side, the library running on a dedicated processor could receive the message, copy it into non-contiguous buffers, free the allocated memory, update the status, and free other receive resources without waiting for an explicit call from the application to peform these operations.

The Intel Paragon and TeraFLOPS machines have multiprocessor nodes for this purpose, and some networking hardware such as Myrinet [1] also have a dedicated message co-processor which could possibly be used in this manner.

Design for Thread Support

One way to take advantage of the available processing resources on a compute node for the purposes described above is through the use of threads. The MPI standard does not explicitly address threads, but the MPI model does not preclude them. The MPI-2 standard contains a model for how threads can be used safely within an MPI application. The model essentially recommends using a communicator per thread.

MPICH is not currently designed to support the use of threads within the device layers. There are some device implementations of MPICH which are thread-safe, where manipulations of globals structures occur within critical regions surrounded by a semaphore. There is also a device implementation for Windows NT which uses multiple threads to handle the queueing and dequeueing of messages [11]. It is expected that future releases of MPICH will be designed for platforms which support some of the basic features available in a thread library such as those defined by the POSIX standard [6].

In anticipation of a multithreaded MPI library, the Sandia implementation was analyzed in order to determine its appropriateness for such a model. The initial implementation uses a single portal to receive MPI messages with the context (communicator) identifier encoded in the matchbits. This approach was taken because portals were viewed as a limited resource since there are only 64 allocated to each application. However, portal numbers are the primary method of message selection and can be used to provide a safe message passing space. A more natural use of portals in MPI is to map a single portal to a single communicator and increase the number of portals available to an application.

Another implementation of MPI was developed which uses this approach. Portals provide a system level context for messages that reduces the amount of global state that needs to be maintained for each communicator. Essentially, messages received on different portals can be received and processed more independently. Reducing the amount of dependence and state reduces the need for critical sections of code which must run atomically. While the first implementation could have been made thread-safe, the second implementation takes better advantage of the inherent Puma message passing model.

However, there are still many issues left to resolve regarding the allocation of message passing resources in this implementation. For example, the number of posted receives for any communicator is fixed at library initialization time. It is likely that the number of posted receives will vary widely for different communicators, especially "hidden" communicators used for collective operations. Likewise, the amount of buffer space needed to hold unexpected messages is fixed at initialization time and is likely to vary considerably for different communicators.

An interesting further question involves determining how to make best use of a threaded message passing library. In some cases, applications have seen reduced performance when naively using threaded utilities. Future research will have to focus on providing thread use guidelines as well as on providing thread capabilities.

Beyond Sends and Receives

Collective Operations

The initial implementation of the MPI collective operations were simply layered on top of MPI peer communcation, which is the default for MPICH. There are several Puma core collective routines which are built directly on top of portals and are optimized for message size and number of participating processes. These collective routines were designed specifically to be a basis for implementing the MPI collective functions. The MPI implementation takes advantage of the Puma collective routines wherever possible, defaulting back to the MPICH layered ones where there is no Puma equivalent operation. A more complete description of the implementation of Puma collective operations can be found in [2].

In modifying MPICH to use these native collective operations, several shortcomings of MPI were identified. First, MPI collective operations are more cooperative than collective. Even though every process that participates in a collective operation must call the same function, the parameters to some functions are not guaranteed to be valid in all processes. That is, there are some collective operations which must ignore certain parameters which are only guaranteed to be usable by the root process. For example, the receive buffer in the MPI_GATHER operation is root significant. If this was not the case, this buffer could be used on the non-root nodes as scratch space for intermediate communication. Likewise, the datatype arguments need not be the same in every participating process. Some operations can involve data which is contiguous on some participating process but non-contiguous on others.

One-Sided Communications

The MPI-2 standard contains a definition of one-sided communications, which differ from point-to-point operations in that the origin process specifies both the sending *and* receiving parameters and the target process need not be explicity involved in the operation. The definition contains functions for data movement operations such as put and get, which are already available on shared memory architectures such as SGI and distributed shared memory Cray T3 platforms.

Portals are ideal for such operations. Because the kernel is responsible for depositing messages, the process receiving the data need not make an explicit call to receive a particular message. Likewise, read memory portals can be set up so that the kernel is responsible for responding to a read memory request. An initial proposal for one-sided communications as it appeared in the March 4, 1996 version of the document was implemented on Puma portals to be used in a code which was being ported from the Cray T3D. A more complete description of that implementation of one-sided communications can be found in [2].

One-sided communications provides the opportunity to get the most performance from Puma Portals, but also exhibits some of the weaknesses. For contiguous datatypes, the put operation provides the opportunity to achieve the greatest bandwidth and lowest latency communications using Portals. However, non-contiguous datatypes expose a weakness of Puma, which is, in part, due to the incomplete implementation of combined block memory descriptors.

MPI two-sided communication requires message headers to be retained. Because any type of MPI send operation can be matched with any receive, MPI point-to-point semantics mandate that some type of protocol information accompany every message so that the receiver can be informed of any action it must perform to complete the operation. In Puma, this information is contained in a message header which must be retained so that the protocol can be identified.

Thus, the Puma implementation of MPI sends and receives had to use the more expensive independent block memory descriptors rather than the single block memory descriptor(in figure 2 the difference is about 13%). The independent block descriptor is more expensive not only because it does the actual header copying but because it does an extra address validation and updates pointers for potential multiple blocks.

Unlike point-to-point communications, there is no protocol information associated with one-sided operations and thus no header information need be retained. The faster single block memory descriptor can now be utilized.

In contrast, one-sided operations with non-contiguous datatypes can require the lowest bandwidth and highest latency communications mechanisms in Puma. Since non-contiguous data can be packed on the send side and unpacked on the receive side for point-to-point messages, the overhead only involves memory-to-memory copies. For one sided, there is no opportunity to unpack data at the receive side, since the receiver need not be involved in the transfer at all.

Therefore, the sender needs to insure that non-contiguous data arrives at the receiver at the correct offsets. The only current way to insure proper offsets is to send each individual block of data as a separate message. The receive portal must be configured to be sender-managed, where an offset into the memory descriptor can be specified in the incoming message. Sending each block as a separate message involves an expensive trap to the kernel for each message. For large numbers of small blocks of data, this overhead can be extremely large.

Although a combined block memory descriptor will help to alleviate some of the overhead involved in performing a gather send operation, there are still complexities to overcome at the receive side. For a put operation, the layout of the data at the target process may only exist at the origin process. That is, the portal on the receiving process must be configured according to the layout described by the sender. This implies that the receiving portal must somehow be configured by the sender. One possible method of allowing a sender to configure a portal at the receiver is to open up another portal over the portal to be configured. This problem also exists for get operations.

Combined Block Issues

In order for the combined block memory descriptor to be useful, it must support both gather sends and scatter receives. Support for scatter receives has been implemented in a development version of the Puma kernel, but support for gather sends is more complex. When sending a message, the Puma kernel validates that the message data is within user space by inspecting the user-supplied buffer and length. Once the address and length have been validated, a message header containing the length of the data is sent to the destination. The destination uses the message length to determine how much data to receive off the mesh. Attempting to read any more or any less than the correct amount of data off of the mesh can lead to subsequent message loss or to the possible lock up of the mesh.

For a gather send, the kernel has to calculate the length of the message based on user-supplied address-length pairs. Once the header has been sent, the sending kernel must send the correct amount of data.

Because the Puma kernel is static in size, it can only validate a limited number of address-length pairs at a time. Should any of these validations fail, the kernel is left to fill up the rest of the message with invalid bytes. There is no way to inform the receiving side that the message contains invalid data, and no way to prevent the receiving kernel from depositing the entire message into the receiving portal.

The result of such a corrupted gather send can be that a remote node's memory is corrupted without any warning, notice, or means of recovery. In contrast, if the memory associated with a receive portal is corrupted by the local node then the kernel can recognize it as an invalid receive portal and discard the incoming message. In order to make the gather send similarly robust, the portal structures would have to include an interface to inform the user that a portal contains bad data. Adding such an interface would require radical changes to the current implementation of Puma.

Real-Time Channels

Early versions of the MPI-2 document contained a proposal for real-time communication. This proposal was moved into the MPI Journal of Development and is now being developed as part of the MPI Real-Time Forum. The proposal for real-time communication is channel-based. A channel is a unidirectional message passing construct that allows for optimizations and guarantees unavailable in current point-to-point communications.

A channel offers the opportunity to exploit persistent communication where there is a one-time cost in choosing buffering schemes, protocols, algorithms, etc., which can be set up at initialization time. Channels are ideal for real-time message passing where guarantees on deadlock avoidance, bandwidth, and buffer space are critical.

Although MPI does provide persistent communication constructs, it allows a persistent send to match any receive and a persistent receive to match any send, eliminating any opportunity to negotiate any parameters *a priori*. The inital setup of MPI-2 channels is a collective operation over a communicator where a quality of service can be guaranteed. All of the overhead in channel setup is moved into the initialization so that the send and receive operations can proceed as quickly as possible.

There is also a proposal for buffering on real-time channels, initially submitted by Sandia, to provide the capability for the MPI system to have some control over the buffering that occurs on a channel. This proposal provides for the application to give a list of buffers to the MPI layer, rather than binding a single buffer to a channel. This buffering scheme allows for 'zero' sided communications in an event-driven paradigm.

Real-time channels have not been implemented on portals, but the experience with the different buffering schemes of of portal memory descriptors has provided valuable input to the buffering proposal submitted to the MPI/RT Forum.

Validating MPI

In preparation to release the original MPI implementation for use on the TeraFLOPS machine, Intel ran the Sandia implementation through its MPI validation test suite. The validation of the MPI library discovered several problems areas. Because error checking is a quality of implementation issue, there were several places in the device independent layers of MPICH where error checking was non-existent or incorrect. While some of this was due to inconsistencies or ambiguities in the standard, much was due to the lack of a complete set of extensive stress tests. Most problems were corrected and the fixes were passed on to the maintainers of MPICH.

Most of the portal-specific problems were with the implementation of collective communications. There is a very specific set of requirements that an application must adhere to in order to use the MPI collectives built on the Puma collectives.

The Puma collective operations are not designed to handle non-contiguous datatypes. Any collective operation which uses non-contiguous datatypes will use the collective operations that are layered on MPI point-to-point calls. However, because type arguments need not be the same on all processes, a collective call must be performed first to insure that all participating processes are using contiguous datatypes.

Because some arguments of the MPI collective calls are only significant at the root process, some information is not available to all of the processes participating in a collective call. As such, the implementation of some of the collective operations had to be modified to assume that these root-significant arguments were not available.

The Puma collective operations were not originally implemented to handle multiple outstanding messages on overlapping subgroups. The implementation of Puma collectives had to be redone in order to support multiple collective operations on different communicators efficiently.

Many of the optimizations available by using Puma collective operations were nullified by strict adherence to the standard. However, most applications will be written in such a way that it may be possible for an implementation to perform some optimizations, given some hints from the programmer. If an application has been written such that the arguments to collective calls are known on all processes, only contiguous datatypes are used, and all reduction operations are commutative, then highly tuned collective operations may be used. The current method for communicating such information is through environment variables. The MPI library might then make a run-time decision about which operations to use.

Conclusion

The implementation of MPI on Sandia's large MPP machines served two purposes. First it allowed Sandia's application programmers to migrate codes to the use of portable MPI syntax and to begin to take advantage of advanced MPI features. The speed at which applications have been brought up on all three ASCI architectures (a large IBM SP-based machine at Lawrence Livermore, and a large SGI/Cray Origin-2000-based machine at Los Alamos, as well as the TeraFLOPS machine at Sandia) has been in part due to Sandia's early adoption of MPI.

Secondly, the co-development of MPI and new MPP operating systems allowed each to be designed and tuned to take advantage of each other. While the measurements presented in this paper concentrate on simple send and receive operations an even larger effect is probably to be seen in scalability and robustness. As work continues on both OS development and message passing libraries at Sandia, it is hoped that these benefits can be quantified.

A recent offshot of the MPI and operating system work is the launch of a new program at Sandia called Computational Plant. A goal of computational plant is to explore the possibility of extending MPP techniques into systems built from commodity processor building blocks and newly available system area network products. The lessons learned from the MPI and system software work will be critical in making this new environment work.

References

[1] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] R. Brightwell and L. Shuler. Design and implementation of MPI on Puma portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.

[3] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[5] W. Gropp and R. Lusk. MPICH working note: The second-generation ADI for the MPICH implementation of MPI. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, February 1996.

[6] ISO/IEC. Information Technology – Portable Operating System Interface (POSIX R) – Part 1: System Application: Program Interface (API). Technical report, IEEE/ANSI Standard 1003.1, 1996. Includes threads interface (1003.1c-1995).

[7] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. Communication on the Paragon. In *Proceedings of the Intel Supercomputer Users' Group. 1993 Annual North America Users' Conference.*, pages 117–124, June 1993.

[8] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.

[9] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[10] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[11] Mississippi State University. *MPI on Windows NT*. http://www.erc.msstate.edu/mpi/mpiNT.html.

[12] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

Distribution:

1	MS 0321	W. J. Camp, 9200
1	MS 0439	D. R. Martinez, 9234
1	MS 0441	R. W. Leland, 9226
2	MS 0619	Review and Approval Desk, 12690 For DOE/OSTI
1	MS 0819	J. Peery, 9231
1	MS 0820	P. Yarrington, 9232
1	MS 0841	P. J. Hommert, 9100
5	MS 0899	Technical Library, 4916
1	MS 1109	A. L. Hale, 9224
1	MS 1110	R. C. Allen, Jr., 9205
1	MS 1110	D. S. Greenberg, 9223
1	MS 1110	D. E. Womble, 9222
1	MS 1111	S. S. Dosanjh, 9221
1	MS 1111	G. S. Hefflefinger, 9225
1	MS 9018	Central Technical Files, 8940–2